Lesson 5
TREES

May 30, 2023

Last time, we learned about the technique of **structural induction** for lists, and used it to prove some theorems about functions on lists.

We also learned about **tail recursion**, and how it can help make code more efficient. We used **accumulators** to facilitate the implementation. We also implemented some useful functions on lists.

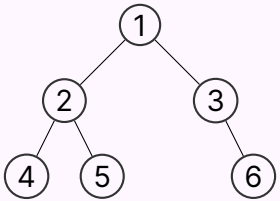
1 - Trees

In computer science, there are only a few fundamental data structures, from which everything else pretty much is derived from.

One of them is lists, which we've already covered. The other is **trees**.

Def A **binary tree** is a data structure of nodes, where each node may have up to two children that are also binary trees.

They look something like this:



SML doesn't have an in-built notion of trees, the same way that it does lists, but we can define our own. We can use a **datatype declaration** to achieve this.

Def A **datatype declaration** is a declaration in SML, which declares a new type.

A datatype declaration for binary trees on integers would look something like this:

```
datatype tree = Empty | Node of tree * int * tree
```

```
datatype tree = Empty | Node of tree * int * tree
```

This datatype declaration does two things. Firstly, it declares a new type, named `tree`. Secondly, it provides that type with two **constructors**, in the same way as the constructors that we saw earlier for lists.

This declaration says that:

- `Empty : tree`
- `Node (L, x, R) : tree` if and only if `L : tree`, `x : int`, and `R : tree`.

In essence, it's saying that a tree only takes two forms, `Empty` or `Node (L, x, R)`. A tree is one of those two things, and **no more**.

Let's see some examples of trees, and how they would be pictured visually[1]:

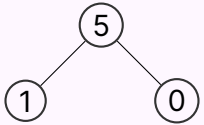
```
Empty : tree
```

(nothing pictured)

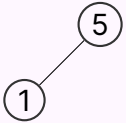
```
Node(Empty, 150, Empty) : tree
```

[1] It can be confusing to type out a tree in text, so if you need help, you can refer to this excellent SML tree converter, which lets you generate SML text for trees from pictures and pictures from SML text, made by previous head TA Sue Lee.

```
Node(Node(Empty, 1, Empty), 5, Node(Empty, 0, Empty)) : tree
```




```
Node(Node(Empty, 1, Empty), 5, Empty) : tree
```



The syntax is important to get used to.

Of our two constructors for the `tree` datatype, only one takes an argument. The other is `Empty`, which is called a **constant constructor**. It requires no arguments in order to be a value of type `tree`.

The constructor that takes an argument is `Node`, which takes in a tuple of type `tree * int * tree`. This type uses the `tree` type we are in the middle of defining!

This is allowed, and called a **recursive type**. Datatypes are allowed to be recursive, in essence saying that the `tree` type can be built out of other `tree`s, with the `Node` constructor.

We can write functions on trees using pattern matching, in the same way that we write functions on lists.

For instance, let's write a function which sums all the nodes of a tree:

```
fun treesum (Empty : tree) : int = 0
  | treesum (Node (L, x, R)) = treesum L + x + treesum R
```

When writing recursive functions on trees, we need two recursive calls instead of one for lists!

Let's write a simple function for computing the depth of a tree.

```
fun max (x : int, y : int) : int =
  if x < y then x else y

fun depth (Empty : tree) : int = 0
  | depth (Node (L, x, R)) = 1 + max (depth L, depth R)
```

Check your understanding Why was this function not written by just directly comparing `depth L` and `depth R`?

2 - Tree Traversals

Sometimes, we are interested in converting data between lists and trees.

In particular, we might be interested in quantities such as determining the number of nodes in a tree. We could, of course, roll our own function to do so:

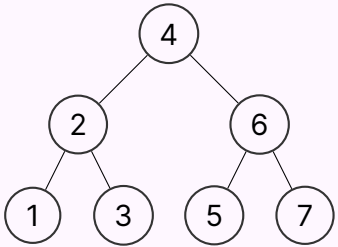
```
fun count (Empty : tree) : int = 0
  | count (Node (L, x, R)) = count L + 1 + count R
```

But this is kind of boilerplate code, and we might want to avoid having to write this!

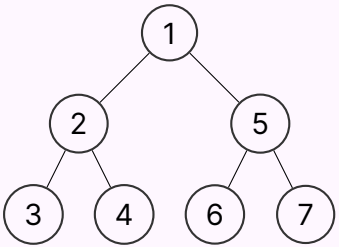
We already have a function for counting the number of elements in a *list*, however.

What if we could turn a tree into a list? We need a function of type `tree -> int list`, which defines a kind of **traversal**.

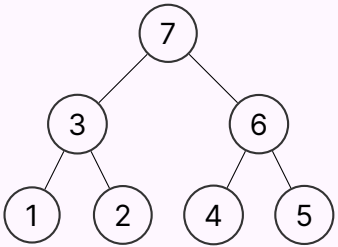
Def A **tree traversal** is a particular kind of way to visit the elements in a tree. They come in three main kinds, **preorder**, **postorder**, and **inorder**.



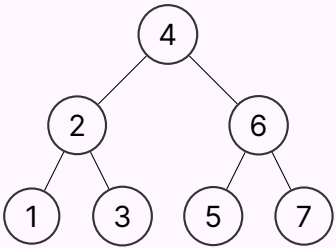
Inorder



Preorder

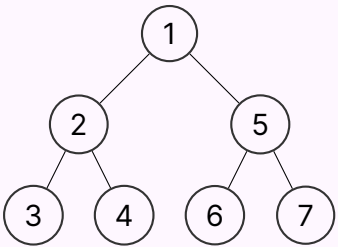


Postorder



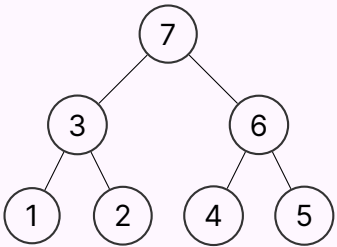
Inorder

left-root-right



Preorder

root-left-right



Postorder

left-right-root

There are three primary kinds of tree traversals, which are characterized by the order in which they choose to visit the left subtree, right subtree, and root. We will usually be concerned with the first two kinds.

The trees are enumerated according to the order in which they are visited, in each traversal.

Tree traversal can be implemented in SML very simply.

```
fun inord (Empty : tree) : int list = []
  | inord (Node (L, x, R)) = (inord L) @ (x :: inord R)

fun preord (Empty : tree) : int list = []
  | preord (Node (L, x, R)) = x :: ((preord L) @ (preord R))
```

Think about how these functions are implemented, with respect to the recursive leap of faith! For `inord`, we can be assured that it follows the left-root-right order, because recursively `inord L` will visit the entire left subtree in an inorder manner, and so will `inord R`. All that remains is to put the pieces together in the right order.

3 - Structural Induction on Trees

Now that we've introduced trees, we are still interested in proving mathematical guarantees on our code!

We will be able to do this for trees in the same flavor as we were able to do for lists.

In the same way that natural numbers are built from other natural numbers, and lists are made of other lists, trees are made from other trees! This means that they admit a principle of structural induction.

Def The principle of **structural induction on trees** is as follows:

Let P be a theorem on values `v : tree`. We would like to show that, for all `v : tree`, $P($`v`$)$ holds.

It suffices to show that:

- $P($`Empty`$)$ holds
- Assuming that $P($`L`$)$ and $P($`R`$)$ hold, for some `L, R : tree`, show that $P($`Node(L, x, R)`$)$ holds, for an arbitrary `x : int`.

Key Fact This means that, when proving a theorem on trees, you get two inductive hypotheses!

Consider the following few functions:

```
fun inord (Empty : tree) : int list = []
  | inord (Node (L, x, R)) = (inord L) @ (x :: inord R)

fun treeSum (Empty : tree) : int = 0
  | treeSum (Node (L, x, R)) = treeSum L + x + treeSum R

fun listSum ([] : int list) : int = 0
  | listSum (x::xs) = x + listSum xs
```

We want to show that both functions `treeSum` and `listSum` do essentially the same thing, when using `inord` to convert between lists and trees.

Thm. For all `T : tree`, `treeSum T` $\cong$ `listSum (inord T)`

Thm. For all values `T : tree`, `treeSum T` $\cong$ `listSum (inord T)`

Lemma 1 For all values `L1, L2 : int list`,
`listSum (L1 @ L2)` $\cong$ `listSum L1 + listSum L2`

Lemma 2 `inord` is total

We proceed by **structural induction** on `T : tree`.

BC `T = Empty`

Let's step the LHS first:

`treeSum Empty` $\cong$ `0` (clause 1 of `treeSum`)

Now the RHS:

`listSum (inord Empty)` $\cong$ `listSum []` (clause 1 of `inord`)

$\cong$ `0` (clause 1 of `listSum`)

IH1 Assume that, for some `L : tree`, `treeSum L` ≅ `listSum (inord L)`.

IH2 Assume that, for some `R : tree`, `treeSum R` ≅ `listSum (inord R)`.

IS Case: `T = Node(L, x, R)`, for some `x : int`. Let's show that `treeSum (Node (L, x, R))` ≅ `listSum (inord (Node (L, x, R)))`.

LHS:

`treeSum (Node (L, x, R))` ≅ `treeSum L + x + treeSum R`

(clause 2 of `treeSum`)

Well, it's not actually clear how to proceed. Let's start from the other side.

RHS:

```
listSum (inord (Node (L, x, R)))
≅ listSum ((inord L) @ (x :: inord R))
```
(clause 2 of `inord`)

Are we stuck here, too? Well, remember we have a lemma! Let's inspect the form of it:

Lemma 1 For all values `L1, L2 : int list`,
`listSum (L1 @ L2) ≅ listSum L1 + listSum L2`
It almost fits!

We can almost use the lemma here. The lemma only applies to expressions of the form `listSum (L1 @ L2)`, where `L1` and `L2` are values.

This is almost what we need! We have `listSum ((inord L) @ (x :: inord R))`. We can see that this looks like the expression `listSum (e1 @ e2)`, where `e1` is `inord L`, and `e2` is `x :: inord R`.

The only way to proceed is to somehow show that `inord L` and `x :: inord R` are values. This would follow quickly if we knew that `inord` was total.

Oh wait, that's one of the lemmas we were given.

This is a general phenomenon I will term **stepping through theorem values**.

In general, you might have a theorem you know, but which only applies to values.

For instance, it might be that for all values `v : int`, `id v` $\cong$ `v`.

You have to be very careful! **This is not the same thing as saying that** `id e` $\cong$ `e`**, for an arbitrary expression** `e`**!**[2]

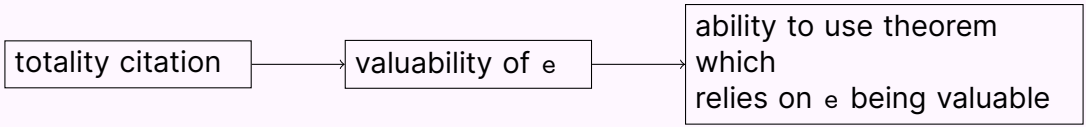
If I wanted to use this theorem to say that `id (length [])` $\cong$ `length []`, I need to show that `length []` **eventually reduces to a value**, i.e. is valuable.

The easiest way to do this is to show that `length` is total. **We use totality as a tool to get at valuability**, so that we can apply theorems that step through values.

[2] Of course, this is not to say that that is a *false* statement. The difference is that when you are making a step, you have to have the **correct** justification. Although you can think about it and conclude that `id e` $\cong$ `e` for an arbitrary expression, **you would have to prove it here**, since it doesn't follow directly from the theorem.

This begins a chapter in your life known as **totality citations**, which are justifications you make in a proof when stepping through theorem values, where you cite a function's totality to get at an expression's valuability.

It is very important to remember the reasoning behind this! If you mindlessly cite totality, you will lose points. The ultimate goal is to use totality citations to conclude an expression's valuability, to use a theorem which relies on a particular expression being valuable.



Back to the proof.

RHS:

```
listSum (inord (Node (L, x, R)))
≅ listSum ((inord L) @ (x :: inord R))                (clause 2 of inord)
≅ listSum (inord L) + listSum (x :: inord R)          (lemma 1, lemma 2)
≅ listSum (inord L) + x + listSum (inord R)
                                          (clause 2 of listSum, lemma 2)
≅ treeSum L + x + treeSum R                   (inductive hypothesis, twice)
```

Now we match the left-hand side, so we can conclude the equivalence.

So by the principal of structural induction on trees, we have proven that for all values `T : tree`, `treeSum T` ≅ `listSum (inord T)`.

4 - Datatypes

Now that we've seen trees, let's return to lists for a bit.

We've seen that we can extract the first element of a list easily by using pattern matching. It's not as clear what to do, however, to obtain the last element in the list!

Let's write a recursive `last` function to achieve this.

```
fun last ([] : int list) : int = (* ??? *)
```

We find that we immediately encounter a problem, however.

Before we even write the recursive case, we have to write the case for the empty list.

What is the last element in the empty list? We want to return an `int`, so it could be `0`, or `1`, or something arbitrary.

The bigger issue is that it simply doesn't make sense to return an integer for this function. We want to return something else, to distinguish the cases of "found 0" and "found nothing".

There are quite often times where we want to write a function, where some inputs do not have a well-defined answer.

We've seen this with the `div` and `fact` functions, where `0` and negative numbers cause an exception to be raised, and infinite looping, respectively.

Sometimes, we don't want such things to happen, however. We would prefer to return a value, which is a more predictable and safe behavior.

To facilitate this, we have the `option` **type constructor**.

A **type constructor** is something which makes a type out of other types.

We were brief about this previously, but that is exactly what `list` is! Out of the types `int`, `bool`, and `int list`, we can make the types `int list`, `bool list`, and `int list list`.

Similarly, we will be able to construct the types `int option`, `string option`, and `int list option`.

Def For any type `t`, there is a type `t option`, which describes a value that is *possibly* a value of type `t`.

The type `t option` has the following constructors:

- `NONE`, which is a constant constructor of no arguments
- `SOME : t -> t option`, which is a constructor that takes a single argument of type `t`.

So for instance, here are some examples of options:

- `SOME 5 : int option`
- `SOME [] : int list option`
- `NONE : int option`
- `NONE : bool option`

Let's rewrite `last` with this knowledge!

```
fun last ([] : int list) : int option = NONE
  | last [x] = SOME x
  | last (x::xs) = last xs
```

We could have raised an exception, but this behavior gives more agency to the caller of the function, in case they want to do something when the list has no output value.

So far, we've seen a few examples of type which have different **variants**, or kinds of constructors that make up the values of the type.

Lists, for instance, can be `[]` or `x::xs`, trees can be `Empty` or `Node(L, x, R)`, and options can be `NONE` or `SOME x`.

These are all known as **variant types**, or **sum types**, and can be defined using the `datatype` keyword! They define all the forms that values of a type can take, and *no more*.

```
datatype int_option = NONE | SOME of int

datatype int_list = [] | :: of int * int list
```

Defining types that fit the shape of the problem is one of the strongest aspects of a functional programming language. These types are known as **algebraic datatypes**.

For instance, suppose we are interested in an ordering function on integers, which has type `int * int -> order`. What should we define the type of `order` to be?

There are three possibilities. The first is less than the second, they are equal, or the first is greater than the second. `bool` will not suffice here!

We could use the description of their relationship as the output value. We can use a **type alias** to alias the name `order` to be the same as the name `string`.

Then, we define our `compare` function:

```
type order = string

fun compare (x : int, y : int) : order =
  if x < y then
    "less"
  else if x = y then
    "equal"
  else
    "greater"
```

This approach is needlessly fragile, however. What does the code look like for a consumer of this function?

```
case compare (x, y) of
  "LESS" => (* code for the less case *)
| "EQUAL" => (* code for the equal case *)
| "GREATER" => (* code for the greater case *)
| _ => (* shouldn't be possible??? *)
```

When casing on the result of `compare`, the compiler doesn't know that any case other than `"LESS"`, `"EQUAL"`, and `"GREATER"` is impossible! This necessitates a redundant extra case.

What's another issue with the above code?

The point is that, although it is *possible* to have `order` be the same as `string`, it comes with a great deal of flaws.

Programming isn't about the possibility of the solution, it's about finding the *best* solution. It's an inherently linguistic process, and we're interested in having good grammatical structure.

We can use datatype declarations to solve this problem in a better way.

Let's define our own `datatype` that fully captures the cases that we are interested in.

```
datatype order = LESS | EQUAL | GREATER

fun compare (x : int, y : int) : order =
  if x < y then
    LESS
  else if x = y then
    EQUAL
  else
    GREATER
```

Now, downstream consumers of this function can write the following code:

```
case compare (x, y) of
  LESS => (* code for the less case *)
| EQUAL => (* code for the equal case *)
| GREATER => (* code for the greater case *)
```

There's no need to have an extra redundant case, and now the compiler will warn if you misspell one of the variants.

This leads to much more polished, streamlined code! Remember the mantra, **Types Guide Structure**. We use the power of types to structure our solution to a problem.

5 - Type Casting

Problems come in all shapes and sizes. The great strength of algebraic datatypes is in being able to craft a type representation that exactly describes your problem.

For instance, suppose we have a class of people. Some people are employed, some people are students, and some people are unemployed.

We might be interested in a couple of things:

- the person's name
- the person's paycheck, if employed, and tuition, if a student
- the company an employed person works for
- the amount of courses a student is taking
- days since last employed, if unemployed (if ever)
- title of last job, if unemployed (if ever)

In service of this, we might define a following datatype which describes a person:

```
datatype class = Employed | Student | Unemployed
```

and the following type which describes the above information:

```
type person_info =
  string (* name *)
  * real (* positive paycheck, if employed. negative
            tuition, if a student, and 0 if unemployed *)
  * string option (* company name, if employed, o.w. NONE *)
  * int option    (* # of courses, if student, o.w. NONE *)
  * int option    (* days since employment, o.w. NONE *)
  * string option (* last job if unemployed, o.w. NONE *)
```

We would then say that a person was a tuple of a `class` and `person_info`.

This is very clearly disgusting. There are floating invariants everywhere!

We have to make sure that people who are employed don't have a number of courses, or a number of days unemployed, or a negative paycheck!

In addition, we might be interested in applying a totally necessary tuition increase. Let's write that function:

```
fun newTuition (_, tuition, _, _, _, _) = tuition * 1.04
```

Except now, this function could possibly run on someone who is just employed or unemployed. Something really bad could happen because this function was written too generically, because the types were all wrong!

Can we do better?

Let's try isolating the parts of each person that are relevant, into the `class` type.

```
type class =
    Employed of real * string
      (* paycheck and company name *)
  | Student of real * int
      (* tuition and # of courses *)
  | Unemployed of (int * string) option
      (* days unemployed and last employer *)

type person = string * class
```

We see that every person has a name, so `string` can be factored out. We can isolate things like company name and number of courses to specific cases, and eliminate the `real` from unemployed people entirely.

For unemployed people who have had previous jobs, we notice that days unemployed and last employer should only be set if both are set, so we fold them both into a single tuple, and make it an `option`, if the person was never employed.

This is a phenomenon I call **type casting**. This is not anything to do with the typical definition of type-casting, involving free conversion of values of one type to another, which does not exist in SML.

Def Similarly to how a blacksmith casts metals to fit a mold of what they need, **type casting** is the art of designing types in a way that they fit the specification of the problem at hand.

An imperative programmer tries to fit square pegs into round holes. A functional programmer can *make* a peg to fit any hole that they need.

Thank you!